I217: Functional Programming

7. Multisets

Kazuhiro Ogata

Roadmap

- Multisets
- Another Implementation of the Mutex Protocol Simulator

Multisets

Collections such that multiple occurrences of elements are permitted and the order in which elements are enumerated is irrelevant

{0,1,2,3} {0,1,0,2,1,3} {0,0,1,1,2,3} {3,2,1,0}

{0,1,2,3} is the same as {3,2,1,0}
but different from {0,1,0,2,1,3}

{0,1,0,2,1,3} is the same as {0,0,1,1,2,3}

Some operator attributes, such as assoc and comm, make it possible to expresses multisets in CafeOBJ.

Multisets

```
mod! MULTISET(E :: TRIV) {  [Elt.E < MSet]      An element is also a singleton multiset.
  op emp : -> MSet {constr} .   The empty multiset
  op _ _ : MSet MSet -> MSet {constr assoc comm id: emp} .
  op if_then{_}else{_} : Bool MSet MSet -> MSet .
  vars MS1 MS2 : MSet .
  eq if true then {MS1} else {MS2} = MS1 .
  eq if true then {MS1} else {MS2} = MS2 .
}
```

$(e_1\ e_2)\ e_3$ is the same as $e_1\ (e_2\ e_3)$ because of **assoc**(iativity).

$e_1\ e_2$ is the same as $e_2\ e_1$ because of **comm**(utativity).

ms emp is the same as *ms* and so is emp *ms* because of **id:** emp.

Multisets

```
open MULTISET(NAT) .
  red (1 2) 3 == 1 (2 3) .
  red 1 2 == 2 1 .
  red 1 1 1 2 2 == 1 2 1 1 2 .
  red emp 1 emp 2 emp 1 emp 1 emp 2 emp .
  red emp .
  red emp emp .
close
```


Multisets

Let us consider multisets of (Qid,Nat)-pairs and a function update that takes such a multiset, a Qid i and a Nat n, and modifies the multiset such that if the multiset contains (i,m), then (i,m) is changed to (i,n).

```
mod! QID-NAT-PAIR principal-sort QNPair {
  pr(QID)  pr(NAT)
  [QNPair]
  op (_,_) : Qid Nat -> QNPair {constr} .
}
mod! TEST {  pr(MULTISET(QID-NAT-PAIR))
  op update : MSet Qid Nat -> MSet .
  vars I J : Qid .  vars N N2 : Nat .  var MS : MSet .
  eq update(emp,J,N2) = emp .
  eq update((I,N) MS,J,N2)
     = if I == J then {(I,N2) MS} else {(I,N) update(MS,J,N2)} .
}
```

Multisets

open TEST .
  red update(('a,1) ('b,2) ('a,3),'a,99) . ➡ ('b , 2) ('a , 3) ('a , 99)
  red update(('a,3) ('b,2) ('a,1),'a,99) . ➡ ('b , 2) ('a , 1) ('a , 99)
close

('a,1) ('b,2) ('a,3) is the same as ('a,3) ('b,2) ('a,1)

But, the results are different.

Something wrong!

There are two pairs ('a,1) and ('a,3) in the multiset ('a,1) ('b,2) ('a,3) (or ('a,3) ('b,2) ('a,1)) to which update can be made. The first reduction chose ('a,1) and the second chose ('a,3), leading to the different results.

Multisets

One remedy for avoiding such different results: if a multiset *ms* of (Qid,Nat)-pairs contain more than one pair whose first element is *i*, then update(*ms*,*i*,*n*) always just becomes *ms*.

update(('a,1) ('b,2) ('a,3),'a,99) ➡ ('a,1) ('b,2) ('a,3)

update(('a,3) ('b,2) ('a,1),'a,99) ➡ ('a,1) ('b,2) ('a,3)

update(('a,1) ('b,2) ('c,3),'a,99) ➡ ('a,99) ('b,2) ('c,3)

update(('c,3) ('a,1) ('b,2),'a,99) ➡ ('a,99) ('b,2) ('c,3)

update(('a,1) ('b,2) ('c,3),'d,99) ➡ ('a,1) ('b,2) ('c,3)

Multisets

Equations can have conditions that are Boolean terms. Such equations are called *conditional equations*.

ceq *LeftTerm* = *RightTerm* **if** *Condition* .

The lest sort of *LeftTerm* is a sort of *RightTerms*.

If *Condition* holds, then *LeftTerm* equals *RightTerm*.

To be able to use it as a conditional rewrite rule, in addition to that *LeftTerm* must not be a single variable and each variable in *RightTerm* must appear in *LeftTerm*, each variable in *Condition* must appear in *LeftTerm* as well.

If σ(*Condition*) reduces to true, then σ(*LeftTerm*) (in a ground term) is a redex and can be replaced with the conract σ(*RightTerm*).

Multisets

Please see the Appendices for BOOL-IF and NAT-IF.

```
mod! TEST2 {
  pr(BOOL-IF)  pr(NAT-IF)
  pr(MULTISET(QID-NAT-PAIR))
  vars I J : Qid .  vars N N2 : Nat .  var MS : MSet .
  -- #
  op # : MSet Qid -> Nat .
  eq #(emp,J) = 0 .
  eq #((I,N) MS,J) = if I == J then {1 + #(MS,J)} else {#(MS,J)} .
```

Given a multiset *ms* and a Qid *i*, it counts the number of pairs (*j*, *n*) in *ms* such that *j* = *i*.

Multisets

```
-- isValid
op isValid : MSet -> Bool .
eq isValid(emp) = true .
eq isValid((I,N) MS)
  = if #(MS,I) > 0 then {false} else {isValid(MS)} .
```

Given a multiset *ms*, it checks if there exists a Qid *i* such that *ms* contains more than one pair whose first element is *i*. If so, it returns false and otherwise, it returns true.

Multisets

```
-- update
op update : MSet Qid Nat -> MSet .
ceq update((I,N) MS,I,N2) = (I,N2) MS if isValid((I,N) MS) .
ceq update(MS,J,N2) = MS if (not isValid(MS)) or #(MS,J) == 0 .
}
```

Given a multiset *ms*, a Qid *i* and a Nat *n*, if *ms* contains exactly one pair whose first element is *i*, then update(*ms*,*i*,*n*) becomes *ms* such that the pair changes to (*i*,*n*), and otherwise it becomes *ms*.

Note that update((I,N) MS,I,N2) does not mean that (I,N) appears as the first element in (I,N) MS, but it just means that (I,N) appears somewhere in (I,N) MS because the juxtaposition operator _ _ is associative and commutative.

Multisets

```
open TEST2 .
  red update(('a,1) ('b,2) ('a,3),'a,99) .   ⇨  ('a,1) ('b,2) ('a,3)
  red update(('a,3) ('b,2) ('a,1),'a,99) .   ⇨  ('a,1) ('b,2) ('a,3)
  red update(('a,1) ('b,2) ('c,3),'a,99) .   ⇨  ('a,99) ('b,2) ('c,3)
  red update(('c,3) ('b,2) ('a,1),'a,99) .   ⇨  ('a,99) ('b,2) ('c,3)
  red update(('a,1) ('b,2) ('c,3),'d,99) .   ⇨  ('a,1) ('b,2) ('c,3)
close
```


Another Implementation of the Mutex Protocol Simulator

We have implemented a Mutex protocol simulator in which states of the Mutex protocol are expressed as tuples, such as

(locked: false, pc1: rs, pc2: rs)

We will revise the simulator such that states of the Mutex protocol are expressed as multisets, such as

(locked: false) (pc[t1]: rs) (pc[t2]: rs)

It is an exercise to consider the pros and cons of each implementation.

Another Implementation of the Mutex Protocol Simulator

Two threads t1 and t2 participate in the protocol:

Loop: "Remainder Section"
rs: **while** *locked* = true {}
ms: *locked* := true;
"Critical Section"
cs: *locked* := false;

Initially, each thread is at rs and *locked* is false.

Another Implementation of the Mutex Protocol Simulator

Nothing is changed for TID and LOC.

```
mod! TID {
 [Tid]
 ops t1 t2 : -> Tid {constr} .
 op if_then{_}else{_} : Bool Tid Tid -> Tid .
 vars T1 T2 : Tid .
 eq if true then {T1} else {T2} = T1 .
 eq if false then {T1} else {T2} = T2 .
}

mod! LOC {
 [Loc]
 ops rs ms cs : -> Loc {constr} .
}
```

Another Implementation of the Mutex Protocol Simulator

```
mod! OCOM { pr(TID) pr(LOC)
  [OCom]
  op (locked:_) : Bool -> OCom {constr} .
  op (pc[_]:_) : Tid Loc -> OCom {constr} .
}
```

(locked: b) and (pc[t]: l) are called *observable components*.
(locked: b) is called a locked observable component, and (pc[t]: l) is called a pc or pc[t] observable component.
(locked: b) means that the value stored in *locked* is b, and (pc[t]: l) means that the location of the thread t is l.
A state is expressed as a multiset of observable components.

(locked: b) (pc[t1]: l_1) (pc[t2]: l_2)

Another Implementation of the Mutex Protocol Simulator

```
mod! STATE principal-sort State {
  pr(OCOM) pr(NAT-IF) pr(BOOL-IF)
  [OCom < State]
  op void : -> State {constr} .
  op _ _ : State State -> State {constr assoc comm id: void} .
  --
  vars T T2 : Tid .
  vars B B2 : Bool .
  var S : State .
  vars L L2 : Loc .
```

States are expressed as multisets of observable components.

Another Implementation of the Mutex Protocol Simulator

```
-- #
op # : State Tid -> Nat .
eq #(void,T) = 0 .
eq #((locked: B) S,T) = #(S,T) .
eq #((pc[T2]: L) S,T)
  = if T2 == T then {1 + #(S,T)} else {#(S,T)} .
```

Given a state *s* expressed as a multiset of observable components and a Tid *t*, it counts the number of pc[*t*] observable components in *s*.

Another Implementation of the Mutex Protocol Simulator

```
-- isValid
op isValid : State -> Bool .
eq isValid(void) = false .
eq isValid((locked: B)) = false .
eq isValid((pc[T]: L)) = false .
eq isValid((pc[T]: L) (locked: B)) = true .
eq isValid((locked: B) (locked: B2) S) = false .
eq isValid((pc[T]: L) (pc[T2]: L2) S)
  = if #((pc[T2]: L2) S,T) > 0 then {false} else {isValid((pc[T2]: L2) S)}
```

(pc[t1]: l_1) (pc[t2]: l_2)

(locked: b) (pc[t1]: l_1) (pc[t2]: l_2) (pc[t1]: l_3)

(locked: b) (locked: b) (pc[t1]: l_1) (pc[t2]: l_2)

(locked: b)

If a state *s* contains exactly one locked observable component and at least one pc observable component and for each thread *t* at most one pc[*t*] observable component, then it returns true, and otherwise it returns false.

Any terms of State such that isValid is false do not express states of the protocol adequately.

Another Implementation of the Mutex Protocol Simulator

```
  -- #cs
  op #cs : State -> Nat .
  eq #cs(void) = 0 .
  eq #cs((locked: B) S) = #cs(S) .
  eq #cs((pc[T2]: rs) S) = #cs(S) .
  eq #cs((pc[T2]: ms) S) = #cs(S) .
  eq #cs((pc[T2]: cs) S) = 1 + #cs(S) .
}
```

It counts the number of pc observable components in a given state such that its location is cs.

This is used to express the Mutex property.

Another Implementation of the Mutex Protocol Simulator

```
mod! FMUTEX {  pr(STATE)
  op trans : State Tid -> State .
  var T : Tid .  var S : State .  var B : Bool .
  ceq trans((pc[T]: rs) (locked: true) S,T) = (pc[T]: rs) (locked: true) S
    if isValid((pc[T]: rs) (locked: true) S) .
  ceq trans((pc[T]: rs) (locked: false) S,T) = (pc[T]: ms) (locked: false) S
    if isValid((pc[T]: rs) (locked: false) S) .
  ceq trans((pc[T]: ms) (locked: B) S,T) = (pc[T]: cs) (locked: true) S
    if isValid((pc[T]: ms) (locked: B) S) .
  ceq trans((pc[T]: cs) (locked: B) S,T) = (pc[T]: rs) (locked: false) S
    if isValid((pc[T]: cs) (locked: B) S) .
  ceq trans(S,T) = S if (not isValid(S)) or (not #(S,T) > 0) .
}
```

If a given state is valid and a given thread appears in the state, a state transition is carried out. Otherwise, nothing changes.

Another Implementation of the Mutex Protocol Simulator

```
mod! COMP {
  pr(INF-LIST(STATE) * {sort InfList -> Comp, sort List -> FComp} )
}

mod! SCHED {
  pr(NAT)
  pr(INF-LIST(TID) * {sort InfList -> Sched} )
  op sched : Nat -> Sched .
  var N : Nat .
  eq sched(N)
     = if 2 divides N
        then {t1 | sched(N quo 2)}
        else {t2 | sched(N quo 2)} .
}
```

Please see the Appendices for INF-LIST.

Nothing is changed for TID and LOC.

Another Implementation of the Mutex Protocol Simulator

```
mod! SIM {  pr(FMUTEX)  pr(COMP)  pr(SCHED)
  op sim : State Nat -> Comp .
  op sub-sim : State Sched -> Comp .
  op sim-check : State Nat Nat -> FComp .
  op sub-sim-check : State Sched Nat -> FComp .
  op mutex : State -> Bool .
  var S : State .  vars N D : Nat .  var NzD : NzNat .
  var T : Tid . var TIL : Sched .
```

Another Implementation of the Mutex Protocol Simulator

```
  eq sim(S,N) = sub-sim(S,sched(N)) .
  eq sub-sim(S,T | TIL) = S | sub-sim(trans(S,T),TIL) .
  eq sim-check(S,N,D) = sub-sim-check(S,sched(N),D) .
  eq sub-sim-check(S,T | TIL,0) = S | nil .
  eq sub-sim-check(S,T | TIL,NzD)
     = if mutex(S)
       then {S | sub-sim-check(trans(S,T),TIL,p NzD)}
       else {S | nil} .
  eq mutex(S) = #cs(S) < 2 .
}
```

Only the equation for mutex has been changed.

Another Implementation of the Mutex Protocol Simulator

```
open SIM .
  red take(sim((locked: false) (pc[t1]: rs) (pc[t2]: rs),123),10) .
  red take(sim((locked: false) (pc[t1]: rs) (pc[t2]: rs),1234),10) .
  red take(sim((locked: false) (pc[t1]: rs) (pc[t2]: rs),12345),10) .
  red sim-check((locked: false) (pc[t1]: rs) (pc[t2]: rs),123,10) .
  red sim-check((locked: false) (pc[t1]: rs) (pc[t2]: rs),1234,10) .
  red sim-check((locked: false) (pc[t1]: rs) (pc[t2]: rs),12345,10) .
close
```

Exercises

1. Write all programs in the slides and feed them into the CafeOBJ system. Moreover, write some more test code and do some more testing for the programs.

2. Revise the simulator (including the version in which the Mutex property is checked) so that it can deal with the case in which there are four threads.

3. Make comparison of the simulator implemented in this lecture with the one implemented in the last lecture and describe the pros and cons of each implementation.

Appendices

```
mod! BOOL-IF {
  op if_then{_}else{_} : Bool Bool Bool -> Bool .
  vars B1 B2 : Bool .
  eq if true then {B1} else {B2} = B1 .
  eq if false then {B1} else {B2} = B2 .
}

mod! NAT-IF {
  pr(NAT)
  op if_then{_}else{_} : Bool Nat Nat -> Nat .
  vars N1 N2 : Nat .
  eq if true then {N1} else {N2} = N1 .
  eq if false then {N1} else {N2} = N2 .
}
```

Appendices

```
mod! GLIST(E :: TRIV) {
  [Nil NnList < List]
  op nil : -> Nil {constr} .
  op _|_ : Elt.E List -> List {constr} .
  op if_then{_}else{_} : Bool List List -> List .
  vars L1 L2 : List .
  -- if_then{_}else{_}
  eq if true then {L1} else {L2} = L1 .
  eq if false then {L1} else {L2} = L2 .
}
```


Appendices

```
mod* INF-LIST(E :: TRIV) {  pr(NAT)  pr(GLIST(E))
  [InfList]
  op _|_ : Elt.E InfList -> InfList {strat: (1 0)} .
  op take : InfList Nat -> List .
  op if_then{_}else{_} : Bool InfList InfList -> InfList .
  var X : Elt.E .  vars IL IL2 : InfList .  var NzN : NzNat .
  -- take
  eq take(IL,0) = nil .
  eq take(X | IL, NzN) = X | take(IL,p NzN) .
  -- if_then{_}else{_}
  eq if true then {IL} else {IL2} = IL .
  eq if false then {IL} else {IL2} = IL2 .
}
```